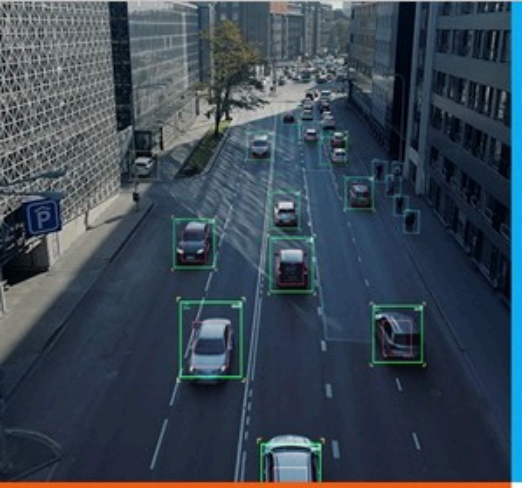


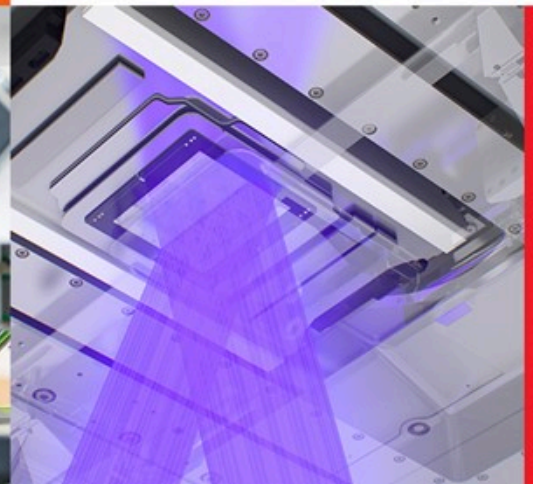
MECHATRONICS

MATHWARE

ASSEMBLY

SIOUX TECHNOLOGIES

ELECTRONICS



SOFTWARE

C++ Training

Reference semantics

Ownership

- Resource management. Owner of a resource
- Responsible for
  -- acquiring the resource
  -- releasing the resource

```
std::string x = "some string";
```

std::string

- Automatic storage variable 'x' manages a heap allocation
- When destructed, the memory used to store the text data is released.
- Owns the memory
  -- Owns the lock
  -- Owns the file handle
  -- owns the thread handle
  -- ...

RAII

Resource acquisition is initialization

- Acquire resource in constructor (initialization)
- Release resource in destructor
- Example: Execute around pointer; Proxy object owns a lock

```
class ThreadsafeVector {
    public:
    using T = std::vector<int>;
    class Proxy {
        public:
            Proxy (T* v) : vector (v) {/*take lock*/}
            T* operator -> () { return vector; }
        ~Proxy () { /*release lock*/}
        private:
            T* vector;
    };
    ThreadsafeVector (T *v) : vector(v) {}
    Proxy operator -> () {
        return Proxy (vector);
    }
    private:
        T* vector;
};
```

- Impossible to access vector without taking the lock.
- Compiler must keep applying operator -> until pointer-like type is encountered

```
int main()
{
    std::vector<int> unsafeVector;
    ThreadsafeVector safeVector(&unsafeVector);
    safeVector->push_back(10);
}
```

Ownership transfer < C++11

```
int main()
{
    std::string s = "12345678910111213141516171819202122232425";
    std::string s2;
    std::cout << "s.data()  " << (void*)s.c_str() << std::endl;
    std::cout << "s2.data() " << (void*)s2.c_str() << std::endl;
    s.swap(s2);
    std::cout << "s.data() " << (void*)s.c_str() << std::endl;
    std::cout << "s2.data() " << (void*)s2.c_str() << std::endl;
    std::cout << "s  " << s << std::endl;
    std::cout << "s2 " << s2 << std::endl;
}
```

```
Program returned: 0
s.data() 0xbfd2b0
s2.data() 0x7fffb187f540
s.data() 0x7fffb187f560
s2.data() 0xbfd2b0
s
S2 123456789… (truncated)
```

Expressing ownership through API

```
class PDF
{
    PDF(std::string title);
    PDF(std::string const& title);
    PDF(char * const title);
};
```

- How to indicate that the PDF class takes ownership of title's data?
- Convention is not compiler enforced.

Move semantics - example

```
struct Blob
{
    Blob(std::size_t size):
        _buffer(size){ }

    void print() const
    {
        std::cout << this << " "<< &_buffer[0] << " size: "<< _buffer.size()<< std::endl;
    }

private:
    std::vector<std::byte> _buffer;
};
```

```
Blob readBlobFromFile(std::string const& path){
    Blob b(1000/*some big file size, assume it reads file at path*/);
    b.print();
    return b;
}
int main(){
    std::multimap<std::string, Blob> storage;
    std::string path = "/usr/bin/cp";
    Blob b = readBlobFromFile(path);
    storage.insert ( {path, b} );
    storage.find(path)->second.print();
    return 1;
}
---
0x7ffe8a0efa90 0x1fac2b0 size: 1000
0x1fadae0 0x1fad6b0 size: 1000
```

Move semantics: What problem ?

```
Blob* readBlobFromFile(std::string const& path)
{
    Blob* b = new Blob(1000/*some big file size*/);
    b->print();
    return b;
}
---
0xfaaeb0 0xfaaed0 size: 1000
0xfaaeb0 0xfaaed0 size: 1000
---
void print() const
{
        std::cout << this << " "<< &_buffer[0] << " size: "<< _buffer.size()<< std::endl;
}
```

That problem !

- Ownership – who frees the allocated memory? Should it be freed?
- Need to read implementation/documentation
- Definition should enforce contract between caller and callee
- A Pointer-type becomes semantically meaningless
- Memory fragmentation

std::move - c++11

```
int main(){
    std::multimap<std::string, Blob> storage;
    std::string path = "/usr/bin/cp";
    Blob b = readBlobFromFile(path);
    storage.insert ( {path, std::move(b)} );
    storage.find(path)->second.print();
    return 1;
}
---
0x7ffcf99442f0 0x886eb0 size: 1000
0x8882f0 0x886eb0 size: 1000
```

std::move

- std::move does not actually move anything
- std::move casts its argument to a temporary
- A temporary has a temporary lifetime from which the resource can be 'moved' from

Possible bugs

```
int main()
{
    std::multimap<std::string, Blob> storage;
    std::string path = "/usr/bin/cp";
    Blob b = readBlobFromFile(path);
    storage.insert ( {path, std::move(b)} );
    storage.find(path)->second.print();
    b.print(); //using a 'moved from' resource
    return 1;
}
---
Program returned: 1
Program stdout
0x7ffd18158c80 0x23f2eb0 size: 1000
0x23f4eb0 0x23f2eb0 size: 1000
0x7ffd18158c80 0 size: 0
```

Definitely bugs

```
void print() const
{
    std::cout << this << " "<< _buffer[0] << " size: "<< _buffer.size()<< std::endl;
}
---
Program returned: 139
Program stdout
0x7ffe8cc72b20 0 size: 1000
0x8bdeb0 0 size: 1000
```

C++ standard: moved from resource

1. **Valid** destructible state
2. What is possible -> implementation defined.
3. Advice: only assume instance can be destructed.

std::move

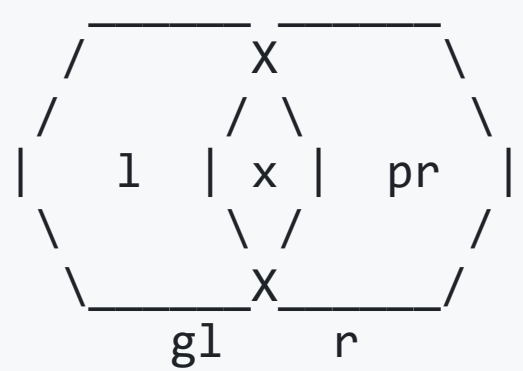
- std::move unconditionally casts input to a movable reference
- std::move does not actually do the move (only casts)

GCC implementation:

```
template<typename T>
std::remove_reference_t<T>&& move(T &&t) noexcept
{
  return static_cast<std::remove_reference_t<T>&&>(t);
}
```

Different types of references

l-value reference, r-value reference, x-values reference



Compiler shall move only when it is unquestionably safe to do so

Special member functions

- Move constructor

```
A(A &&a) = default;
```

- Move assignment operator

```
A& operator==(A &&a) = default;
```

Special member functions

```
struct A
{
  A();
  A(A &&a) noexcept;
  A& operator=(A &&a);
  LargeResource *v;
  std::vector<B> w;
};
A(A &&a) noexcept
{
  v = a.v;
  w = std::move(a.w);
  a.v = nullptr;
};
```

```
#include <iostream>
struct Blob
{
    Blob(){}
    Blob(Blob const& lvalue){std::cout << "lvalue ctor" << std::endl;}
    Blob(Blob && rvalue){std::cout << "rvalue ctor" << std::endl;}
};

int main()
{
    Blob b;
    Blob b2(b);
    Blob b3(std::move(b));
}
---
lvalue ctor
rvalue ctor
```

Converts lvalue ‘b’ to xvalue(rvalue). Allowing compiler to call rvalue constructor when creating b3.

```
struct X
{
    X(std::size_t capacity){
         std::cout << "ctor" << std::endl;
         _container.reserve(capacity);
    }
    X(X const& other) {
        _container  = other._container;
        std::cout << "copy ctor" << std::endl;
    }
    X(X&& other)  {
        _container  = std::move(other._container);
        std::cout << "move ctor" << std::endl;
    }

    std::vector<int> _container;
};
X test(){
    return X{1};
}
int main() {
    X x = test();
}
```

```
int main()
{
    X const& v(test());
    X a(v);
    X b(std::move(v));
}
---
ctor // RVO
copy ctor
move ctor
```

```
int main()
{
    X v(X());
}
---
ctor
move ctor
```

```cpp
class A {};

int main()
{
    std::cout << std::boolalpha;
    std::cout << std::is_lvalue_reference<A>::value << '\n';
    std::cout << std::is_lvalue_reference<A&>::value << '\n';
    std::cout << std::is_lvalue_reference<A&&>::value << '\n';
    std::cout << std::is_lvalue_reference<int>::value << '\n';
    std::cout << std::is_lvalue_reference<int&>::value << '\n';
    std::cout << std::is_lvalue_reference<int&&>::value << '\n';
}
```

Special member functions

- The default move assignment & constructor are generated
  if no copy assignment, constructor or destructor is defined by user
- Note: =default and =delete count as user defined

Exercises

reference_semantics/ex1.cpp

reference_semantics/ex2.cpp

exercises on move semantics/avoiding copies

Move semantics

Perfect forwarding

```
template <typename T, ???>
unique_ptr<T> make_unique(???)
{
  return unique_ptr<T>(new T(???));
}
```

What should we replace the ??? with?

Perfect forwarding

```
template <typename T, typename Arg>
unique_ptr<T> make_unique(Arg && arg)
{
  return unique_ptr<T>(new T(arg));
}
```

One problem, arg is an lvalue now

Perfect forwarding

```
template <typename T, typename Arg>
unique_ptr<T> make_unique(Arg && arg)
{
  return unique_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

Perfect forwarding

```
//l-value overload
template< class T >
constexpr T&& forward( std::remove_reference_t<T>& t ) noexcept;
//r-value overload
template< class T >
constexpr T&& forward( std::remove_reference_t<T>&& t ) noexcept;
```

```
class A;
A a;
auto b = make_unique<A>(a);
```

```
unique_ptr<A> make_unique(A& && arg)
{
  return unique_ptr<A>(new A(std::forward<A&>(arg)));
}
```

```
unique_ptr<A> make_unique(A& && arg)
{
  return unique_ptr<A>(new A(static_cast<A& &&>(arg)));
}
```

```
template <typename T, typename Arg>
unique_ptr<T> make_unique(Arg && arg)
{
  return unique_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

```
class A;
auto a = make_unique<A>(A());
```

```
unique_ptr<A> make_unique(A&& && arg)
{
  return unique_ptr<A>(new A(std::forward<A&&>(arg)));
}
```

```
unique_ptr<A> make_unique(A&& && arg)
{
  return unique_ptr<A>(new A(static_cast<A&& &&>(arg)));
}
```

Overloading a forwarding constructor

```
struct A
{
  A(std::string const& a);         // (1)
  template<typename T> A(T &&t);   // (2)
};

std::string str("str");
A a2(str);                  // calls ctor (2)
                            // because the argument is NOT const
```

Overloading a forwarding constructor

```
struct A
{
  A(const std::string &a);         // (1)
  template<typename T> A(T &&t);   // (2)
};

A a3(a2);     // calls ctor (2)
              // because the argument is NOT const
              // and the default generated copy ctor accepts a “const A&”
```

Guideline: Do not overload templated constructors

Overloading

```
void fun(      A& ) { std::cout << "fun (1)\n"; }
void fun(const A& ) { std::cout << "fun (2)\n"; }
void fun(      A&&) { std::cout << "fun (3)\n"; }
void fun(const A&&) { std::cout << "fun (4)\n"; }
template<typename T> void fun(T &&) { std::cout << "fun (5)\n"; }
template<typename T> void fun(const T &&) { std::cout << "fun (6)\n"; }

int main()
{
  A a{};
  fun(a);
}
```

Overloading

```
void fun(      A& ) { std::cout << "fun (1)\n"; }  (1)
void fun(const A& ) { std::cout << "fun (2)\n"; }  (3)
void fun(      A&&) { std::cout << "fun (3)\n"; }
void fun(const A&&) { std::cout << "fun (4)\n"; }
template<typename T> void fun(T &&) { std::cout << "fun (5)\n"; } (2)
template<typename T> void fun(const T &&) { std::cout << "fun (6)\n"; }

int main()
{
  A a{};
  fun(a);
}
```

Overloading

```
void fun(      A& ) { std::cout << "fun (1)\n"; }
void fun(const A& ) { std::cout << "fun (2)\n"; }
void fun(      A&&) { std::cout << "fun (3)\n"; }
void fun(const A&&) { std::cout << "fun (4)\n"; }
template<typename T> void fun(T &&) { std::cout << "fun (5)\n"; }
template<typename T> void fun(const T &&) { std::cout << "fun (6)\n"; }

int main()
{
  const A a{};
  fun(a);
}
```

```
void fun(       A& ) { std::cout << "fun (1)\n"; }
void fun(const A& ) { std::cout << "fun (2)\n"; }  (1)
void fun(       A&&) { std::cout << "fun (3)\n"; }
void fun(const A&&) { std::cout << "fun (4)\n"; }
template<typename T> void fun(T &&) { std::cout << "fun (5)\n"; }  (2)
template<typename T> void fun(T const &&) { std::cout << "fun (6)\n"; }

int main()
{
  const A a{};
  fun(a);
}
```

```
template<typename T>
void fun(T && t)
{
    std::cout << "fun (5): " << std::is_const_v<std::remove_reference_t<T>>;
}
```

```
void fun(      A& ) { std::cout << "fun (1)\n"; }
void fun(const A& ) { std::cout << "fun (2)\n"; }
void fun(      A&&) { std::cout << "fun (3)\n"; }
void fun(const A&&) { std::cout << "fun (4)\n"; }
template<typename T> void fun(T &&) { std::cout << "fun (5)\n"; }
template<typename T> void fun(const T &&) { std::cout << "fun (6)\n"; }

A get();

int main()
{
  fun(get());
}
```

```
void fun(      A& ) { std::cout << "fun (1)\n"; }
void fun(const A& ) { std::cout << "fun (2)\n"; } (5)
void fun(      A&&) { std::cout << "fun (3)\n"; } (1)
void fun(const A&&) { std::cout << "fun (4)\n"; } (3)
template<typename T> void fun(T &&) { std::cout << "fun (5)\n"; } (2)
template<typename T> void fun(const T &&) { std::cout << "fun (6)\n"; } (4)

A get();

int main()
{
  fun(get());
}
```

Whats wrong here?

```
struct A
{
  template<typename T>
  A(T&& t)
    : m_b(std::move(t))
  {}

  B m_b;
};
```

Invalid move if T is l-value

```
struct A
{
  template<typename T>
  A(T&& t)
    : m_b(std::forward<T>(t))
  {}

  B m_b;
};
```

Whats wrong here?

```
template<typename T>
struct A
{
  A(T&& t)
    : m_b(std::forward<T>(t))
  {}

  B m_b;
};
```

T is a class template

```
template<typename T>
struct A
{
  A(T&& t)
    : m_b(std::forward<T>(t))
  {}

  B m_b;
};
```

exercises

reference_semantics/ex3.cpp

reference_semantics/ex4.cpp

exercises on universal references

Move semantics

Library support

- std::unique_ptr
- std::shared_ptr
- std::weak_ptr

std::unique_ptr

- Expresses single ownership
- Own the pointer or own the resource
- RAII (memory allocation, sockets, files, …)
- NOT limited to memory, can provide custom destructor to deal with other types of resources.

Strategy pattern

- Swap implementation based on some condition

```
#include "sink.h"

int main(){

    Sink defaultLogger(0);
    defaultLogger.write(std::string("Not a sussy log"));

    Sink secretLogger(1);
    secretLogger.write(std::string("I can do my dance like a touchdown"));

    return 1;
}
```

Sink.h

```
#include <memory>
#include <string>
struct Sink{
    Sink(int);
    ~Sink();
    void write(std::string const& in);
    struct Encryptor; //forward declared Encryptor
private:
    std::unique_ptr<Encryptor> pEncryptionImpl;
    void _write(std::string const& in);
};
```

Sink.cpp

```cpp
#include "sink.h"
#include <algorithm>
#include <iostream>
//can be stateful
struct Sink::Encryptor
{
    virtual std::string encrypt(std::string const& in) = 0;
    virtual ~Encryptor(){};
};
struct NullEncrypt: Sink::Encryptor
{
    std::string encrypt(std::string const& in) final{
        return in;
    }
};
struct GigaEncrypt: Sink::Encryptor{
    std::string encrypt(std::string const& in) final{
        std::string _v= in;
        std::reverse(std::begin(_v), std::end(_v));
        return _v;
    }
};
```

Sink.cpp continued

```
//~Sink() needs to be defined in cpp
Sink::~Sink(){
}
Sink::Sink(int encryptionLevel){
    //doesnt have to be invariant
    if(!encryptionLevel){
        pEncryptionImpl = std::make_unique<NullEncrypt>();
    }
    else{
        pEncryptionImpl = std::make_unique<GigaEncrypt>();
    }

}

void Sink::write(std::string const& in){
    std::string encrypted = pEncryptionImpl->encrypt(in);
    _write(encrypted);
}
void Sink::_write(std::string const& in){
    std::cout << in << std::endl;
}
```

- Sink; it writes, it encrypts. Or does it?
- If we want to change the way things are encrypted we don’t have to change sink
- If we want to change how things are written to output, we NEED to change sink. Output strategy?

std::unique_ptrs and C-API

```
#include <memory>
void* create_rpc() {
    return new int();
}
void free_rpc(void* p){
    delete (int*) p;
}
int main()
{
    std::unique_ptr<void, void(*)(void*)> managedRpc(create_rpc(), free_rpc);
    return sizeof(managedRpc); //16 = 2 * 8
}
```

- Size of pointer is increased: 16. 64 bit system, should be 8.
- Cannot inherit from function pointer
- Need a struct with an operator that takes in pointer as parameter, aka lambda

```cpp
#include <memory>
#include <iostream>
void* create_rpc() {
    return new int();
}
void free_rpc(void* p){
    delete (int*) p;
}
int main()
{
    std::unique_ptr<void, void(*)(void*)> managedRpc(create_rpc(), free_rpc);
    std::cout << "function ptr. size: "<< sizeof(managedRpc) << std::endl;

    auto free_rpc_lambda = [](void* p){delete (int*) p;};
    std::unique_ptr<void, decltype(free_rpc_lambda)> managedRpc2(create_rpc());
    std::cout << "lambda. size: "<< sizeof(managedRpc2) << std::endl;

    std::unique_ptr<void, decltype([](void* p){return free_rpc(p);})> managedRpc3(create_rpc());
    std::cout << "lambda. size: "<< sizeof(managedRpc3) << std::endl;
}
---
function ptr. size: 16
lambda. size: 8
lambda. size: 8
```

std::shared_ptr

1. Expresses shared ownership
2. Multiple owners of a resource
3. Reference counted
4. Either strong(std::shared_ptr) or weak ownership (std::weak_ptr)
5. Extra allocation, reference counter is MT safe

```cpp
class CallbackBase
{
public:
    virtual void invoke() = 0;
    virtual ~CallbackBase() = default;
};
class CallbackManager {
public:
    void addCallback(std::weak_ptr<CallbackBase> callback) {
        callbacks.push_back(callback);
    }

    void invokeCallbacks() {
        auto it = callbacks.begin();
        while (it != callbacks.end()) {
            if (auto locked = it->lock()) {
                locked->invoke();  // Invoke the callback
                ++it;              // Move to the next callback
            } else {
                 std::cout << "Callback erased" << std::endl;
                it = callbacks.erase(it);  // Remove expired weak_ptr
            }
        }
    }
private:
    std::vector<std::weak_ptr<CallbackBase>> callbacks;
};
```

```
class MyCallback : public CallbackBase, public std::enable_shared_from_this<MyCallback> {
public:
    void invoke() override {
        std::cout << "Callback invoked!" << std::endl;
    }
    void registerCallback(CallbackManager& manager) {
        manager.addCallback(shared_from_this());
    }
};
int main() {
    CallbackManager manager;
    {
        auto callback1 = std::make_shared<MyCallback>();
        callback1->registerCallback(manager);
        auto callback2 = std::make_shared<MyCallback>();
        callback2->registerCallback(manager);
        manager.invokeCallbacks();
    }
    manager.invokeCallbacks();
    return 0;
}
---
Program returned: 0
Callback invoked!
Callback invoked!
Callback erased
Callback erased
```

Move semantics

- Save time and space by avoiding allocations and expensive copies
- Expressing ownership – resource is being passed around

Elliding pointers

- Pointers can be expensive. chasing pointers. fragmentation. allocations.
- Common pointer avoiding techniques:
    i. Empty base optimization
    ii. Empty string optimization

Empty base optimization

- Hijack some member variable to instantiate a strategy/inject a policy.
- Empty base class (no member vars) does not increase size of derived class.

  gcc:

```
struct _Alloc_hider : _Alloc
{
    _Alloc_hider(_CharT* __dat, const _Alloc& __a) _GLIBCXX_NOEXCEPT
    : _Alloc(__a), _M_p(__dat) { }

    _CharT* _M_p; // The actual data
};
```

```
struct Allocator {
    void allocate(){}
    void deallocate(){}
};
struct MyString {
    char* _M_p;
    Allocator _M_a;
};
struct MyStringSmall {
    struct Hider : Allocator
    {
        char* _M_p;
    };

    Hider _M_hider;
};
int main() {
    std::cout << "MyString: "<< sizeof(MyString) << std::endl;
    std::cout << "MyStringSmall: "<<sizeof(MyStringSmall) << std::endl;
}
---
MyString: 16
MyStringSmall: 8
```

Small buffer optimization

```
template <typename T>
struct CustomAllocator
{
    T* allocate(std::size_t n)
    {
        auto ptr = std::malloc(n * sizeof(T));
        std::cout << "allocate "  << std::hex<< ptr<< std::endl;
        return static_cast<T*>(ptr);
    }
    void deallocate(T* ptr, std::size_t n)
    {
        std::cout << "deallocate " << std::hex<< (void*)ptr<< std::endl;
        std::free(ptr);
    }
};
using MyString = std::basic_string<char, std::char_traits<char>, CustomAllocator<char>>;
int main()
{
    MyString a("large buffer. large buffer. large buffer. large buffer. "),b("small buff");
    std::cout << a << b << std::endl;
}
```

```
Program returned: 0
Program stdout
allocate 0x1967eb0
large buffer. large buffer. large buffer. large buffer. small buff
deallocate 0x1967eb0
```

No heap allocation for 'b'

gcc

```
template<typename _CharT, typename _Traits, typename _Alloc>
template<typename _InIterator>
void basic_string<_CharT, _Traits, _Alloc>::_M_construct(_InIterator __beg, _InIterator __end, std::forward_iterator_tag)
{
    // NB: Not required, but considered best practice.
    if (__gnu_cxx::__is_null_pointer(__beg) && __beg != __end)
        std::__throw_logic_error(__N("basic_string::_M_construct null not valid"));

    size_type __dnew = static_cast<size_type>(std::distance(__beg, __end));

    if (__dnew > size_type(_S_local_capacity))
    {
        _M_data(_M_create(__dnew, size_type(0)));
        _M_capacity(__dnew);
    }

    ...

    _M_set_length(__dnew);
}
```

```
union
{
    _CharT           _M_local_buf[_S_local_capacity + 1];
    size_type        _M_allocated_capacity;
};
```

- _M_local_buf overlaps with _M_allocated_capacity
- reduce memory footprint, no allocation, less fragmentation.
- std::function, std::vector do this.

Which constructor do I use?

- Move / copy constructor, most optimal
- Often you will see code like this:

```
struct X {
    X(MyString const& s):_s(s){}
    MyString _s;
};
int main() {
    MyString b("this is the tale of tony montana");
    X x2(std::move(b));
}
---
allocate 0x1b8eeb0
allocate 0x1b8fef0
deallocate 0x1b8fef0
deallocate 0x1b8eeb0
```

Take a copy, allocate less

```
#include "mystring.h"
struct X
{
    X(MyString s):_s(std::move(s)){}
    MyString _s;
};
int main()
{
    MyString b("this is the tale of tony montana");
    X x2(std::move(b));
}
---
allocate 0x2034eb0
deallocate 0x2034eb0
```

- Copy then move if l-value (you have to copy anyway)
- Move then Move if r-value
- l-value/r-value ctor. No extra move in both cases.
- Justifiable

exercises

reference_semantics/ex5.cpp

Implement std::unique_ptr optionally with EBO

We bring **high-tech** to life